2. Exploring Abstractions: Information Structures

2.1 Introduction

Which comes first, the abstraction or the implementation? There is no fixed answer. Historically abstractions were introduced as ways of simplifying the presentation of implementations and then standardizing implementations. Increasingly, developers are being encouraged to think through the abstractions first, then find appropriate ways to implement them. In this section we will discuss various abstractions for structuring information, which is related to implementation issues of data structures.

We assume that the reader has been exposed to (uni-directional) linked-list data structuring concepts. Linked lists are often presented as a way to represents sequences in computer memory, with the following property:

> Insertion of a new item into a linked list, given a reference to the insertion point, entails at most a fixed number of operations.

Here “fixed” is as opposed to a number of operations that can vary with the length of the sequence. In order to achieve this property, each item in the list is accompanied by a *pointer* to the next item. In the special case of the last item, there is no next, so a special *null pointer* is used that does not point to anything.

After, or along with, the process of demonstrating linked list manipulation, box diagrams are typically illustrated. These show the items in the list in one compartment of a box and the pointer in a second compartment, with an arrow from that compartment to the box containing the next item. A diagonal line is typically used instead of the arrow in the case of a null pointer. Finally, a pointer leading from nowhere indicates the first element of the list.

For example, a sequence of four elements: a, b, c, d would be shown as in the following box diagram.

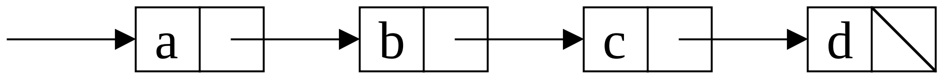


Figure 3: A box diagram for a linked list of four elements

It is worth mentioning that box diagrams are themselves abstractions of data inside the computer. In order to implement the concept of “pointing”, we typically appeal to an addressing or indexing mechanism, wherein each potential box has an implied numeric address or index. (Addressing can be regarded as a special case of indexing in which the entire memory is indexed.) The underlying implementation of the box diagram abstraction could be shown as in Figure 4, where the numbers at the left are the indices of the boxes. We use an index value of –1 to represent the null pointer. The empty boxes are currently unused. With this representation, we also need to keep track of the first element, which in this case is the one at index 0. Note that there can be many representations for a single abstraction; this is a common phenomenon.

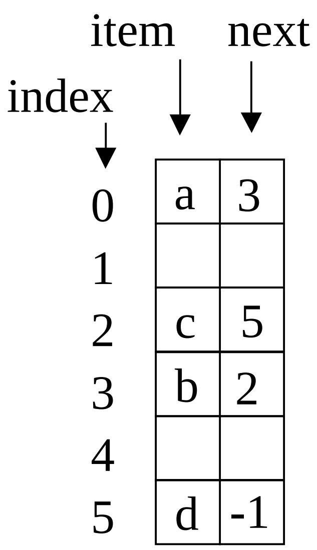


Figure 4: A representation, using indices, for a linked list of four elements

At the representation level, in order to insert a new element, we simply obtain a new box for it and adjust the structure accordingly. For example, since the box at index 1 is unused, we can use it to hold the new element. Suppose the element is e anl it is to be inserted after b. The new representation picture would be as shown in Figure 5

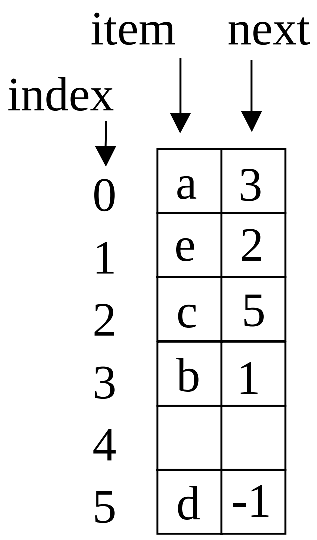


Figure 5: A representation for the modified linked list, of five elements

Returning to the abstract level, we can show the sequence of modifications for inserting the new element as in Figure 6.

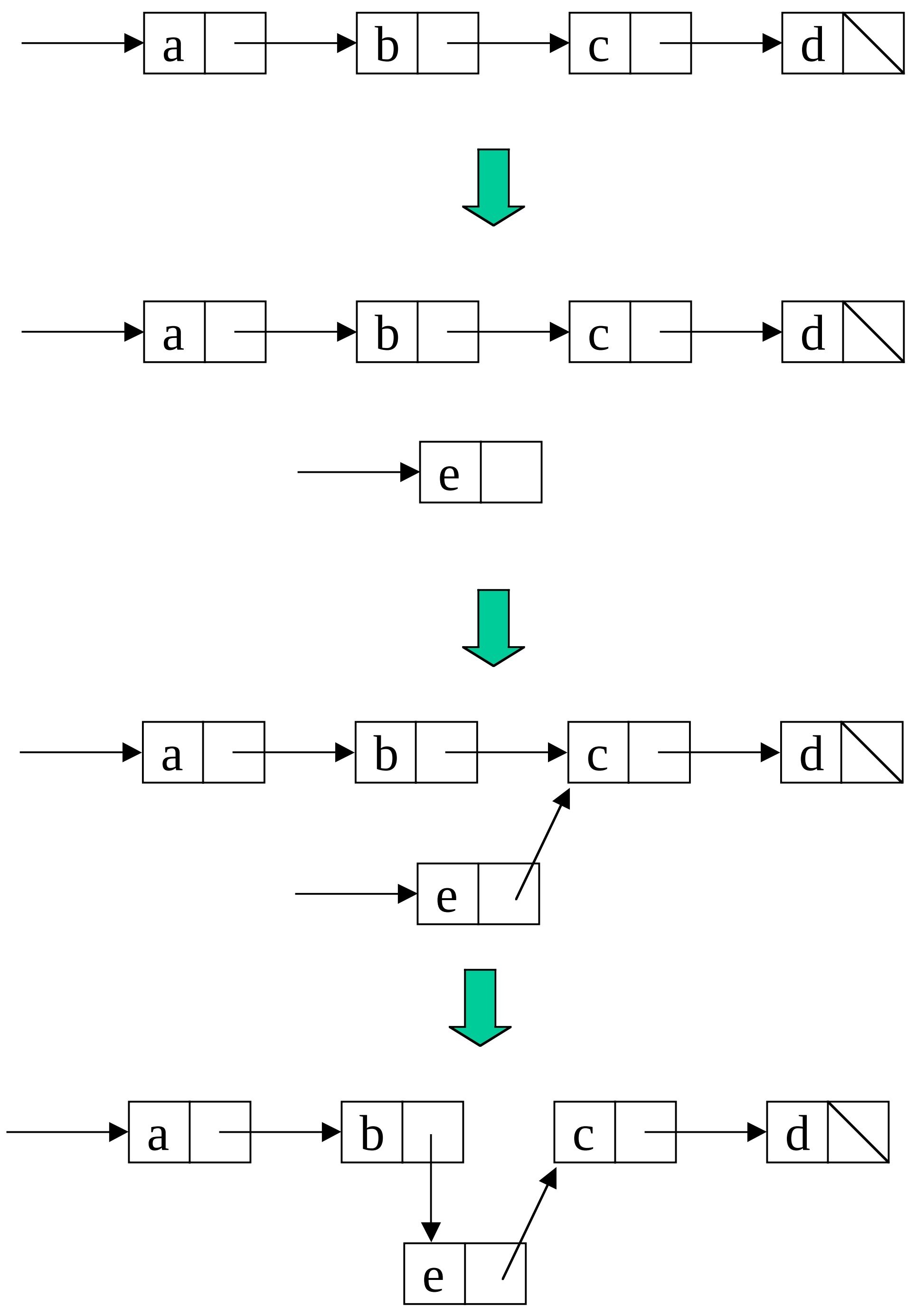


Figure 6: Sequence of changes in a box diagram to represent insertion. The arrows show the changes to the state of the list.

From the third state in the sequence of changes, we can observe something interesting: in this state, two pointers point at the box containing item, rather than one. This suggests an interesting possibility: that parts of lists can be *shared* between two or more lists. There is a fundamental split in list processing philosophies which has to do with whether such sharing is exploited or discouraged. This text refers to the philosophy in which lists are frequently shared as *open* lists, and the opposite philosophy as *closed* lists. Chapter 5 gives more details on the distinctions between the two.

The present chapter is going to focus on open lists almost exclusively. The next figure shows two lists sharing a common "tail". The positive aspect of such sharing is that the space saved for the stem is used only once even though the effect, from the viewpoint of each list, is that each enjoys equal use of the tail. The negative aspect is that we must be extremely careful about any modifications that take place in the tail; if the user of one list modifies the tail, then the user of the other list "feels" the modification. This might be unintended. Because it is hard to manage these types of changes, it is typical to *forbid modifications* to lists in the open list model.

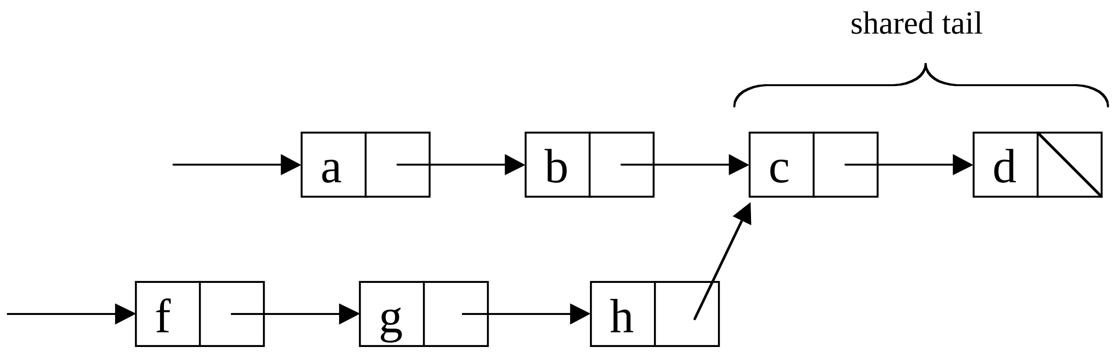


Figure 7: Two lists with a shared tail

One might wonder, if modifications to lists are forbidden in the open list model, how does anything get done? The answer may be surprising: *only create new lists.* In other words, to achieve a list that is similar to an existing list, create a new list so modified. This might seem incredibly wasteful, but it is one of the major philosophies for dealing with lists. Moreover, we still have the possibility of tail sharing, which can overcome a lot of the waste, if we do things correctly. Since we agree to modify the structure of a list, we can share a tail of it freely among an arbitrary number of lists. In fact, the shared tails need not all be the same, as shown in the Figure 8.

Notice that it only makes sense to share tails, not "heads", and to do so it simply suffices to have access to a pointer to the start of the sub-list to be shared.

So far we have seen the box abstraction for open lists, and not shown its lower-level implementation in a programming language. What we will do next is develop a textual abstraction for dealing with open lists. Then we will use *it* as the *implementation* for some higher level abstractions. This illustrates how abstractions can be stacked into *levels*, with one level of abstraction providing the implementation for the next higher level.

Our approach derives structurally from languages such as Lisp, Prolog, and their various descendants. An interpreter for our language, called *rex*, is available on common platforms such as UNIX® and Windows, so that the specifications can be understood and tried interactively.

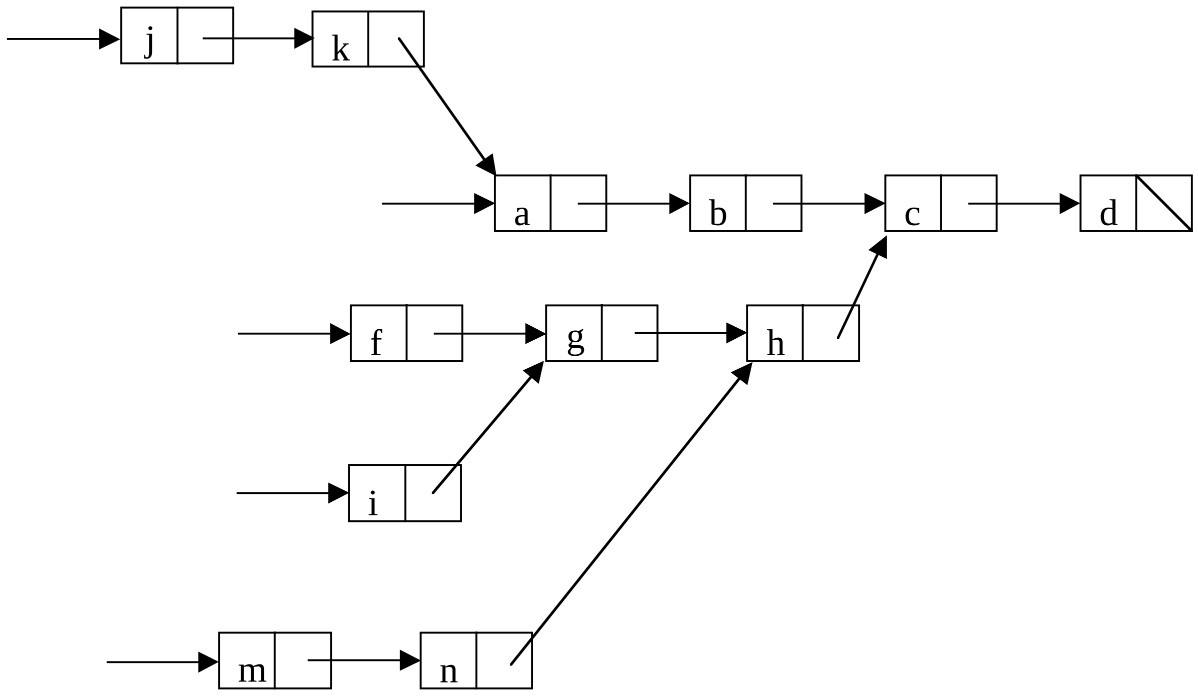


Figure 8: Multiple tail-sharing

Despite the fact that we express examples in our own language, the reader should be reassured that *the medium is not the message*. That is, we are not trying to promote a specific language; the ideas and concepts are of main interest. From this neutrality comes the possibility of using the ideas in many different languages. A few examples of mapping into specific languages will be shown in this book. For example, in addition to rex, we will show a Java implementation of the same ideas, so that the programming concepts will carry forth to Java. We contend that is easier to see the concepts for the first time in a language such as rex, since there is less syntactic clutter.

We should also comment on information structures vs. *data structures*. The latter term connotes structures built in computer memory out of blocks of memory and references or pointers. While an information structure can be implemented in terms of a data structure, the idea of information structures attempts to suppress specific lower-level implementation considerations. Put another way, an information structure is an *abstraction* of a data structure. There could be several different data structures implementing one information structure. This is not to say that all such data structures would be equally attractive in terms of performance. Depending on the intended set of operations associated with an information structure, one data structure or another might be preferable.

2.2 The Meaning of "Structure"

Computing is expressed in terms of primitive elements, typically numbers and characters, as well as in terms of structures composed of these things. Structures are often built up hierarchically: that is, structures themselves are composed of other structures, which are composed of other structures, and so on. These more complex structures are categorized by their relative complexity: trees, dags (directed acyclic graphs), graphs, and so on. These categories are further refined into sub-categories that relate to particular applications and algorithmic performance (roughly translating into the amount of time a computation takes): binary-search trees, tries, heaps, etc.

When a datum is not intended for further subdivision, it is called *atomic*. The property of being atomic is relative. Just as in physics, atoms can be sub-divided into elementary particles, so in information, atomic units such as numbers and character strings could conceivably be sub-divided. Strings could be divided into their characters. Numbers could, in a sense, be divided by giving a representation for the number using a set of digits. Of course, in the case of numbers, there are many such representations, so the subdivision of numbers into digits is not as natural as for strings into characters.

2.3 Homogeneous List Structures

We begin by describing a simple type of structure: lists in which all elements are of the same type, which are therefore called *homogeneous* lists. However, many of the ideas will also apply to heterogeneous lists as well. To start with, we will use numerals representing numbers as our elements.

A list of items is shown in its entirety by listing the items separated by commas within brackets. For example:

```
[2, 3, 5, 7]
```

is a list of four items. Although the notation we use for lists resembles that commonly used for sets (where curly braces rather than square brackets are used), there are some key differences:

- Order matters for lists, but not for sets.
- Duplication matters for lists, but not for sets.

Thus while the sets {2, 5, 3, 7, 3} and {2, 3, 5, 7} are regarded as equal, the two lists`[2, 5, 3, 7, 3]` and `[2, 3, 5, 7]` are not.

Two lists are defined to be *equal* when they have the same elements in exactly the same order.

2.4 Decomposing a List

Consider describing an algorithm for testing whether two lists are equal, according to the criterion above. If we were to base the algorithm directly on the definition, we might do the following steps:

To test equality of two lists:

- Count the number of items in each list. If the number is different then the two lists aren't equal. Otherwise, proceed.
- Compare the items in each list to each other one by one, checking that they are equal. If any pair is not equal, the two lists aren't equal. Otherwise, proceed.
- Having passed the previous tests, the two lists are equal.

Although this algorithm seems fairly simple, there are some aspects of it that make less than minimal assumptions, for instance:

- It is necessary to know how to *count* the number of items in a list. This requires appealing to the concept of number and counting.
- It is necessary to be able to *sequence* through the two lists.

Let's try to achieve a simpler expression of testing list equality by using the device of list decomposition. To start, we have what we will call the *fundamental list-dichotomy*.

fundamental list-dichotomy:

A list is either:

- *empty*, i.e. has no elements, or
- *non-empty*, i.e. has a first element and a rest

By *rest*, we mean the list consisting of all elements other than the first element. The empty list is shown as `[ ]`. It is possible for the rest of a non-empty list to be empty. Now we can re-cast the equality-testing algorithm using these new ideas:

To test equality of two lists:

a. If both lists are empty, the two lists are equal.
   (proceeding only if one of the lists is not empty.)

b. If one list is empty, the two lists are not equal.
   (proceeding only if both lists are non-empty.)

c. If the first elements of the lists are unequal, then the lists are not equal.
   (proceeding only if the first elements of both are equal.)

d. The answer to the equality of the two lists is the same as the answer to whether the rests of the two lists are equal. This equality test in this box can be used.

Let us try this algorithm on two candidate lists: `[1, 2, 3]` and`[1, 2]`.

Equality of `[1, 2, 3]` and`[1, 2]`:

Case *a*. does not apply, since both lists are non-empty, so we move to case *b*.

Case *b*. does not apply, since neither list is empty, so we move to case *c*.

Case *c*. does not apply, since the first elements are equal, so we move to case d.

Case *d*. says the answer to the equality question is obtained by asking the original question on the rests of the lists: `[2, 3]` and `[2]`.

Equality of `[2, 3]` and `[2]`:

Cases *a*. through *c*. again don't apply.

Case *d*. says the answer to the equality question is obtained by asking the original question on the rests of the lists: `[3]` and `[ ]`.

Equality of `[3]` and `[ ]`:

Case *a*. does not apply.

Case *b*. does apply: the two lists are not equal.

The differences between this algorithm and the first one proposed are that this one does not require a separate counting step; it only requires knowing how to decompose a list by taking its first and rest. Although counting is implied by the second algorithm, it can be "short-circuited" if the two lists are found to be unequal.

In the next chapter we will see how express such algorithms in more succinct terms.

2.5 List Manipulation

Here we introduce the interactive rex system, which will provide a test-bed for dealing with lists. When we type an expression to rex, followed by the semi-colon, rex shows the *value* of the expression. The reason for the semi-colon is that some expressions take more than one line. It would thus not work to use the end-of-line for termination of expressions. Other approaches, such as using a special character such as \ to indicate that the list is continued on the next line, are workable but more error-prone and less aesthetic (lots of \'s traded for one ;). For lists as introduced so far, the value of the list as an expression will be the list itself. Below, the boldface shows what is entered by the user.

```
rex > [2, 3, 5, 7];
[2, 3, 5, 7]

rex > [ ];
[ ]
```

As it turns out, list equality is built into rex. The equality operator is designated ==.

```
rex > [1, 2, 3] == [1, 2];
0
```

The value 0 is used by rex to indicate *false*: that the two lists are not equal. A value of 1 is used to indicate *true*.

```
rex > [ ] == [ ];
1
```

Suppose we wanted to access the first and rest of a list in rex. Two approaches are available: (*i*) use built-in functions *first* and *rest*; (*ii*) use list matching, which will be described in Chapter 4. Here we illustrate (*i*):

```
rex > first([1, 2, 3]);
1

rex > rest([1, 2, 3]);
[2, 3]
```

Correspondence with Box Diagrams

It is worthwhile at this point to interject at this time the correspondence between textual representation and box diagrams. This mainly helps to motivate the way of manipulating lists that will be described presently, especially to justify the choice of primitive functions being used.

Every list in the open list model can be identified with a pointer to the first box in the box representation. The first element of the list is the item in the box and the list of the remaining elements is the pointer. The pointer is the special null pointer if, and only if, the rest of the list is empty. Figure 9 is meant to illustrate this correspondence.

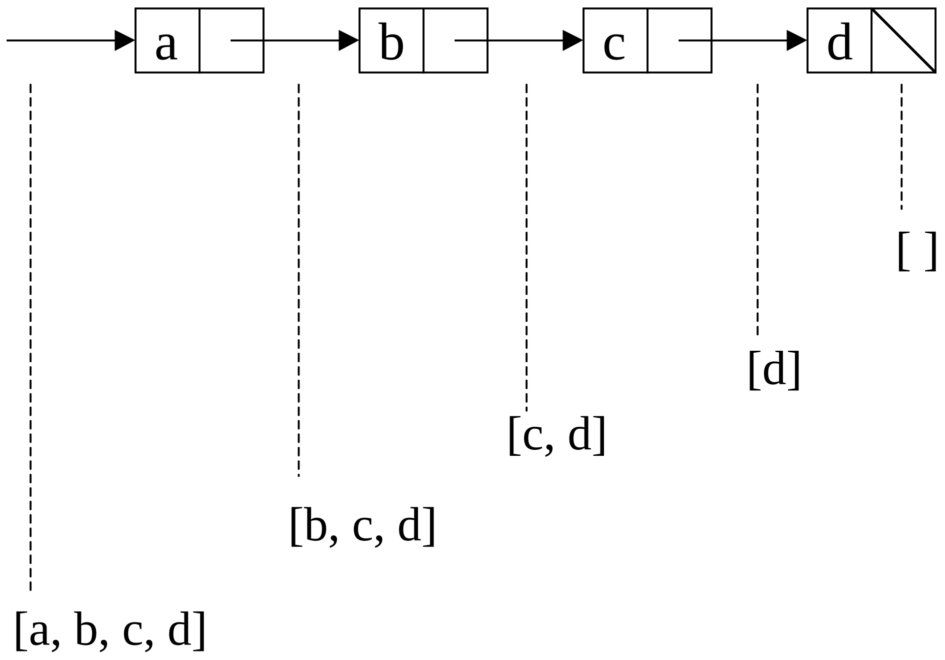


Figure 9: Equating pointers to lists in the open list representation

Intuitively, the `rest` operation is very fast in this representation. It does not involve creating any new list; it only involves obtaining the pointer from the first box in the list.

Identifiers in rex

An *identifier* in rex is a string of symbols that identifies, or stands for, some item such as a list or list element. As with many programming languages, a rex identifier must begin with a letter or an underscore, and can contain letters, digits, and underscores. The following are all examples of identifiers:

```
a     Temp  x1    _token_     move12      _456        ZERO
```

The most basic way to cause an identifier to stand for an item is to equate it to the item using the *define* operator, designated = (recall that == is a different operator used testing equality):

```
rex > a = 2;
1
```

This defines identifier `a` to stand for the number 2. In computer science it is common to say `a` is *bound to* 2 rather than "stands for" 2.

Bindings

If an identifier is bound to something, we say that it is *bound*, and otherwise it is *unbound*. By a *binding*, we mean a pairing of an identifier with its value, as in the binding `["a", 2]`, which is implicitly created inside the system by the above definition. Finally, an *environment* is a set of bindings. In other words, an environment collects together the meanings of a set of symbols.

Above, the reported value of 1 indicates that the definition was successful. We can check that *a* now is bound to 2 by entering the expression a by itself and having rex respond with the value.

```
rex > a;
2
```

On the other hand, if we try this with an identifier that is not bound, we will be informed of this fact:

```
rex > b;
*** warning: unbound symbol b
```

Forming New Lists

The same symbols used to decompose lists in rex can also be used to form new lists. Suppose that `R` is already bound to a list and `F` is an intended first element of a new list. Then `[F | R]` (again read `F` “followed by” `R`) denotes a new list with `F` as its first element and `R` as the rest of the new list:

```
rex > F = 2;
1

rex > R = [3, 4, 5];
1

rex > [F | R];
[2, 3, 4, 5]
```

One might rightfully ask why we would form a list this way instead of just building the final list from the beginning. The reason is that we will have uses for this method of list creation when either `F` or `R` are *previously* bound, such as being bound to data being supplied to a function. This aspect will be used extensively in the chapter on low-level functional programming.

As with our observation about rest being fast, creating a new list in this way (called “consing”, short for “constructing”) is also fast. It only entails getting a new box, initializing its item, and planting a pointer to the rest of the list.

Lists of Other Types of Elements

In general, we might want to create lists of any type of element that can be described. For example, a sentence could be a list of strings:

```
[“This”, “list”, “represents”, “a”, “sentence”]
```

A list can also be of mixed types of elements, called a *heterogeneous* list:

```
[“The”, “price”, “is”, 5, “dollars”]
```

We can get the type of any element in rex by applying built-in function `type` to the argument:

```
rex > type(99);
integer

rex > type("price");
string

rex > type(["price", 99]);
list
```

The types are returned as strings, and normally strings are printed without quotes. There is an option to print them with quotes for development purposes. Notice that giving the type of a list as simply “list” is not specific as to the types of elements. Later we will see how to define another function, `deep_type`, to do this:

```
rex > deep_type(["price", 99]);
[string, integer]
```

Here the deep type of a list is the list of deep types of the individual elements.

Using Lists to Implement Sets

For many computational problems, the ordering of data and the presence of repetitions either are not to be important or are known not to occur at all. Then it is appropriate to think of the data as a *set* rather than a list. Although a list inherently imposes an ordering on its elements, we could simply choose to ignore the ordering in this case. Also, although repetitions are allowed in lists, we could either ignore repetitions in the case of sets or we could take care not to permit repetitions to enter into any lists we create.

Since abstraction has already been introduced as a tool for suppressing irrelevant detail, let us think of sets as a distinct abstraction, with open lists as one way to represent sets. An implementation, then, would involve giving list functions that represent the important functions on sets

For example, suppose we wanted to represent the set function `union` on lists. A use of this function might appear as follows:

```
rex > union([2, 3, 5, 7, 9], [1, 4, 9, 16]);
[2, 3, 5, 7, 1, 4, 9, 16]
```

Above the element occurs in both original sets, but only occurs once in the union. Similarly, we can represent the set function `intersection`:

```
rex > intersection([2, 3, 5, 7, 9], [1, 4, 9, 16]);
[9]
```

This result is correct, since `9` is the only element occurring in both sets. These functions are not built into rex; we will have to learn how to define them.

Representation Invariants

Our discussion of representing sets as lists provides an opportunity to mention an important point regarding representations. Implementing functions on sets represented as lists is more economical (is faster and takes less memory space) if we can assume that the list used to represent the set contains no duplicates. A second reason for maintaining this assumption is that the implementation of certain functions becomes simpler. For example, consider a function `remove_element` that removes a specified element from a set, provided that it occurs there at all. Without the assumption that there are no duplicates, we would need to check the entire list to make sure that all occurrences of the element were removed. On the other hand, if we can assume there are no duplicates, then we can stop when the first occurrence of the element in question has been found.

Assumptions of this form are called *representation invariants*. Their articulation is part of the implementation and each function implemented has the obligation of making sure that the invariant is true for the data representation for the value it returns. At the same time, each function enjoys the use of the assumption to simplify its work, as described in the preceding paragraph.

2.6 Lists of Lists

Lists are not restricted to having numbers as elements. A list can have lists as elements:

```
[ [2, 4, 8, 16], [3, 9, 27], [5, 25], [7] ]
```

The elements of the outer list can themselves be lists.

```
      [ [ [1, 2, 3], [4, 5, 6] ], [ [7, 8, 9], [ ] ] ]
```

Pursuing our sets-implemented-as-lists idea, a list of lists could be used to represent a set of sets. For example, the function `subsets` returns all subsets of its argument, interpreted as a set:

```
  rex> subsets([1, 2, 3]);
  [ [ ], [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3] ]
```

This function is not built in, but we will see how to define it in a later chapter.

Length of Lists

The *length* of a list is the number of elements in the list:

```
      rex > length([2, 4, 6, 8]);
      4
```

When the list in question has lists as elements, *length* refers to the number of elements in the outer or *top-level* list, rather than the number of elements that are not lists:

```
      rex > length([ [2, 4, 8], [3, 9], [5], [ ] ]);
      4
```

The number of elements that are not lists is referred to as the *leafcount*:

```
      rex > leafcount([ [2, 4, 8], [3, 9], [5], [ ] ]);
      6
```

Exercises

1. • Which pairs of lists are equal?

   a. `[1, 2, 3]` *vs*. `[2, 1, 3]`

   b. `[1, 1, 2]` *vs*. `[1, 2]`

   c. `[1, [2, 3] ]` *vs*. `[1, 2, 3]`

   d. `[1, [2, [3] ] ]` *vs*. `[1, 2, 3]`

2. •• Execute the equality testing algorithm on the following two lists: `[1, [2, 3], 4]`, and `[1, [2], [3, 4]]`.

3. ••• As we saw above, although lists are not the same as sets, lists can be used to represent sets. Present an equality testing algorithm for two sets represented as lists.

4. • What is the length of the list `[1, [2, 3], [4, 5, 6]]`? What is its leafcount?

2.7 Binary Relations, Graphs, and Trees

A *binary relation* is a set of ordered pairs. Since both sets and pairs can be represented as lists, a binary relation can be represented as a list of lists. For example, the following list represents the relation *father_of* in a portion of the famous Kennedy family:

```
[     [“Joseph”, “Bobby”],
      [“Joseph”, “Eunice”],
      [“Joseph”, “Jean”],
      [“Joseph”, “Joe Jr.”],
      [“Joseph”, “John”],
      [“Joseph”, “Pat”],
      [“Joseph”, “Ted”]
      [“Bobby”,  “David”],
      [“Bobby”,  “Joe”]
      [“John”,   “Caroline”],
      [“John”,   “John, Jr.”],
      [“Ted”,    “Edward”] ]
```

Here Joseph, for example, is listed as the first element in many of the pairs because he had many children. Notice also that no individual is listed as the second element in more than one pair. This reflects the fact that an individual cannot have more than one father.

Graphical Representation of Binary Relations

Binary relations are sometimes rendered more understandable by depiction as a *directed graph*: the nodes in the graph are items being related; arrows in the graph represent the pairs in the relation. For example, the preceding *father_of* relation would be shown as below:

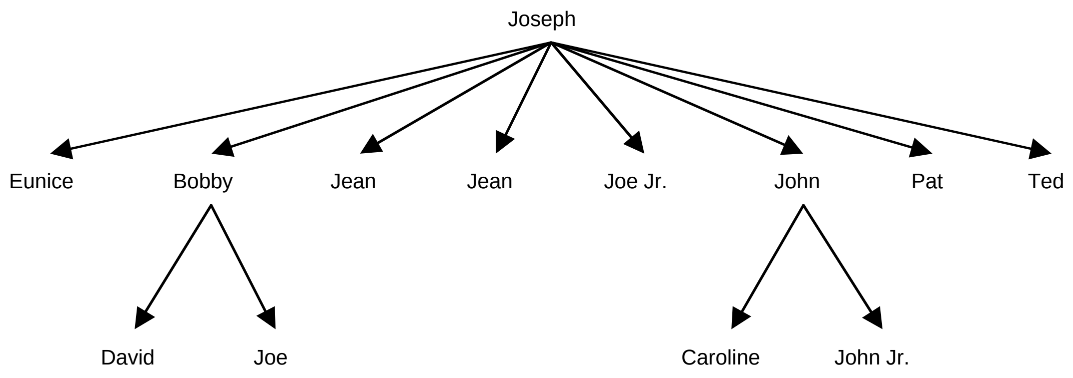


Figure 10: a father_of relation

Notice that each arrow in the directed graph corresponds to one ordered pair in the list. This particular directed graph has a special property that makes it a *tree*. Informally, a graph is a tree when its nodes can be organized hierarchically.

Let's say that node *n* is a *target* of node *m* if there is an arrow from *m* to *n*. (In the above example, the targets happen to represent the children of the individual.) Also, the *target set* of a node *m* is the set of nodes that are targets of node *m*. For example, above the target set of `"Bobby"` is {`"David"`, `"Joe"`}, while the target set of `"Joe"` is the empty set.

Let's say the nodes *reachable* from a node consist of the targets of the node, the targets of those targets, and so on, all combined into a single set. For example, the nodes reachable from `"Joseph"` above are all the nodes other than `"Joseph"`. If the nodes reachable from some node include the node itself, the graph is called *cyclic*. If a graph is not cyclic then it is called *acyclic*. Also, a node that is not a target of any node is called a *root* of the graph. In the example above, there is one root, `"Joseph"`. A node having no targets is called a *leaf*. Above, `"Eunice"`, `"Joe"`, and `"Caroline"` are examples of leaves.

Given these concepts, we can now give a more precise definition of a tree.

A *tree* is a directed graph in which the following three conditions are present:

- The graph is acyclic
- There is exactly one root of the graph.
- The intersection of the target sets of any two different nodes is always the empty set.

If we consider any node in a tree, the nodes reachable from a given node are seen to form a tree in their own right. This is called the *sub-tree* determined by the node as root. For example, above the node `"Bobby"` determines a sub-tree containing the nodes {`"Bobby"`, `"David"`, `"Joe"`} with `"Bobby"` as root.

Hierarchical-List Representation of Trees

A list of pairs, while simple, is not particularly good at allowing us to spot the structure of the tree while looking at the list. A better, although more complicated, representation would be one we call a *hierarchical list*:

To represent a tree as a hierarchical list:

- The root is the first element of the list.
- The remaining elements of the list are the sub-trees of the root, each represented as a hierarchical list.

These rules effectively form a representation invariant.

This particular list representation has an advantage over the previous list-of-pairs representation in that it shows more clearly the *structure* of the tree, *viz*. the children of each parent. For the tree above, a list representation using this scheme would be:

```
[“Joseph”,
   [“Eunice”],
   [“Bobby”,
      [“David”],
      [“Joe”] ],
   [“Jean”],
   [“Joe Jr.”],
   [“John”,
      [“Caroline”],
      [“John Jr.”] ],
   [“Pat”],
   [“Ted”,
      [“Edward”] ] ]
```

Note that in this representation, a leaf will always show as a list of one element, and every such list is a leaf. Also, the empty list only occurs if the entire tree is empty; the empty list representing a sub-tree would not make sense, because it would correspond to a sub-tree with no root.

As a variant on the hierarchical list representation, we may adopt the convention that leaves are not embedded in lists. If we do this for the current example, we would get the representation

```
[“Joseph”,
   “Eunice”,
   [“Bobby”,
      “David”,
      “Joe”],
   “Jean”,
   “Joe Jr.”,
   [“John”,
      “Caroline”,
      “John Jr.”],
   “Pat”,
   [“Ted”,
      “Edward”] ]
```

which has the virtue of being slightly less cluttered in appearance. From such a list we can reconstruct the tree with labels on each node. Accordingly, we refer to this as the *labeled-tree interpretation* of a list.

Another example that can exploit the hierarchical list representation of a tree abstraction is that of a *directory structure* as used in computer operating systems to manage collections of files. A common technique is to organize files in "directories", where each directory has an associated collection of files. A file space of this form can be thought of as a list of lists. Typically directories are not required to contain only files; they can contain other directories, which can contain files or directories, etc. in any mixture. So a user's space might be structured as follows (we won't put string quotes around the names, to avoid clutter; however, if this were rex, we would need to, to avoid confusing the labels with variables):

```
[ home_directory,
  mail,
  [ mail_archive, current_mail, old_mail],
  [ programs,
    [ sorting, quicksort.rex, heapsort.rex],
    [ searching, depth_first.rex, breadth_first.rex]
  ],
  [ games, chess, checkers, tic-tac-toe]
]
```

representing the tree in Figure 11.

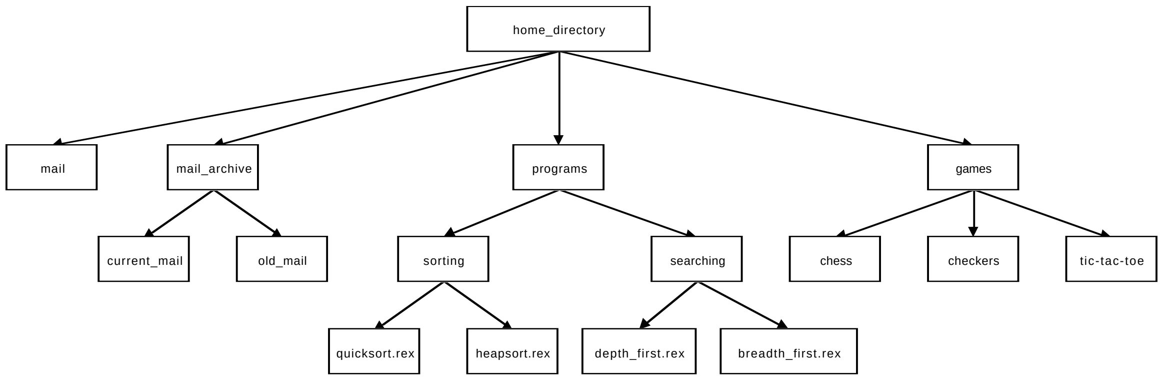


Figure 11: A directory tree

Unlabelled vs. Labeled Trees as Lists

Sometimes it is not necessary to carry information in the non-leaf nodes of the tree. In such cases, we can eliminate this use of the first list element, saving one list element. We call this the *unlabeled-tree interpretation* of the list. The only list elements in such a representation that are not lists themselves are the leaves of the tree. In this variation, the list

```
[a, [b, c], [d, [e, f], g], h]
```

represents the following tree (as is common practice, we often omit the arrow-heads, with the understanding that they all point the same direction, usually downward):

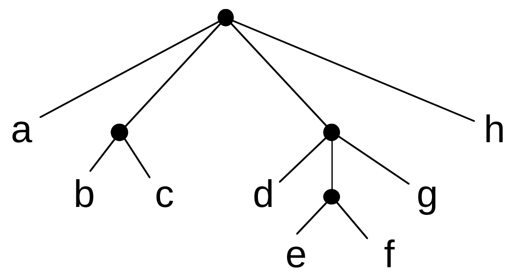


Figure 12: The list `[a, [b, c], [d, [e, f], g], h]` as an unlabeled tree

In the hierarchical list representation described previously, the first element in a list represents the label on a root node. This tree would have a similar, but slightly different shape tree (assuming the convention that we don't make lists out of single leaves):

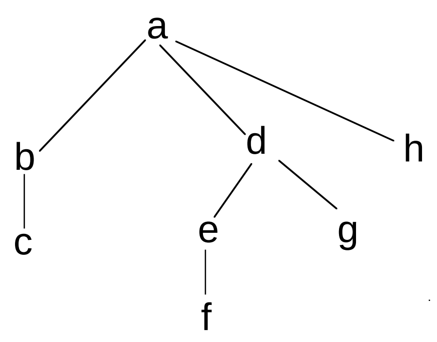


Figure 13: The list `[a, [b, c], [d, [e, f], g], h]` as a labeled tree

Clearly, when we wish to interpret a list as a tree, we need to say which interpretation we are using. We summarize with the following diagrams:

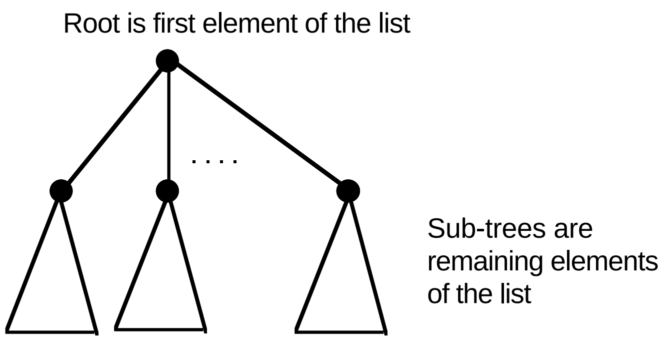

Root is first element of the list
. . . .
Sub-trees are remaining elements of the list

Figure 14: Hierarchical list representation of a labeled tree

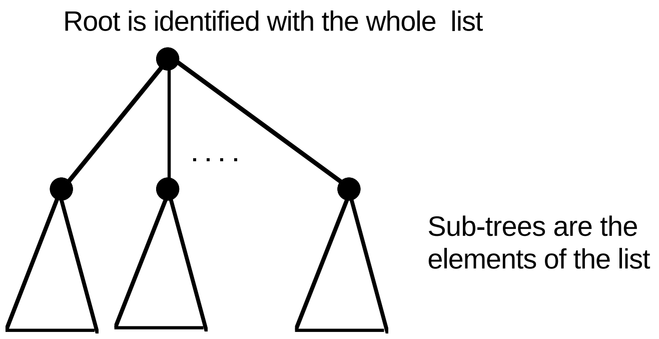

Root is identified with the whole list
. . . .
Sub-trees are the elements of the list

Figure 15: Interpretation of a list as an unlabeled tree

Note that the *empty list* for an unlabeled tree would be the empty tree, or the slightly anomalous "non-leaf" node with no children (since leaves have labels in this model).

Binary Trees

In the *binary-tree representation* of a list, we recall the view of a list as [*First* | *Rest*] where, as usual, *First* is the first element of the list and *Rest* is the list consisting of all but the first element. Then the following rules apply:

***Binary tree representation as lists*:**

- The **empty list** is shown as a leaf [ ].
- An **atom** is shown as a leaf consisting of the atom itself.
- A list [*First* | *Rest*] is shown as a node with two subtrees, one tree corresponding to *First*, the other the tree corresponding to *Rest*.

A representation invariant here is the right-hand branch is always a list, never an atom.

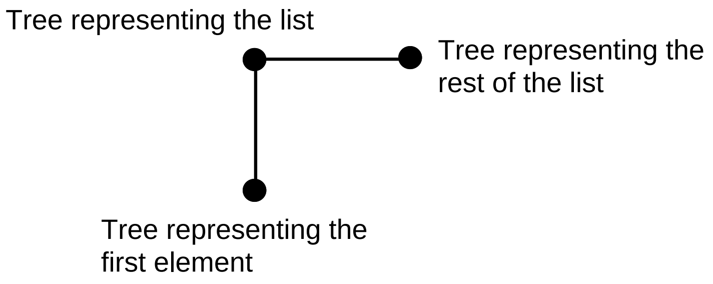

Tree representing the list
Tree representing the rest of the list
Tree representing the first element

Figure 16: Binary tree representation of a list

In particular, a list of n (top-level) elements can easily be drawn as a binary tree by first drawing a "spine" with n sections:

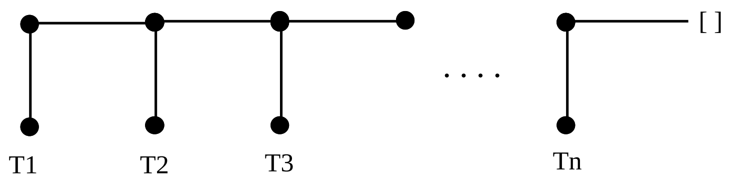


Figure 17: Binary tree representation of a list with top-level elements T1, T2, ..., Tn

The elements on the branches of the spine can then drawn in.

For the preceding example, `[a, [b, c], [ ], [d, [e, f], g], h]`, these rules give us the following tree, where the subtrees of a node are below, and to the right of, the node itself.

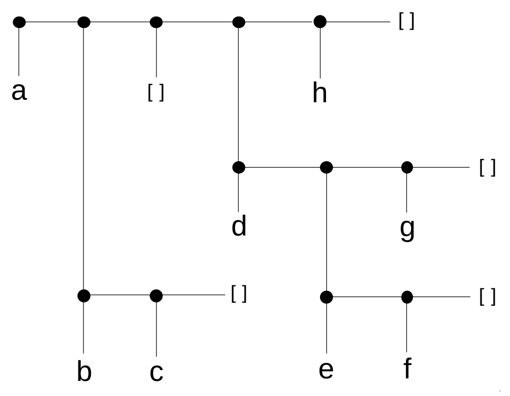


Figure 18: The binary tree representation of a list
`[a, [b, c], [ ], [d, [e, f], g], h]`

Sometimes preferred is the rotated view where both subtrees of a node are below the node. The same tree would appear as follows.

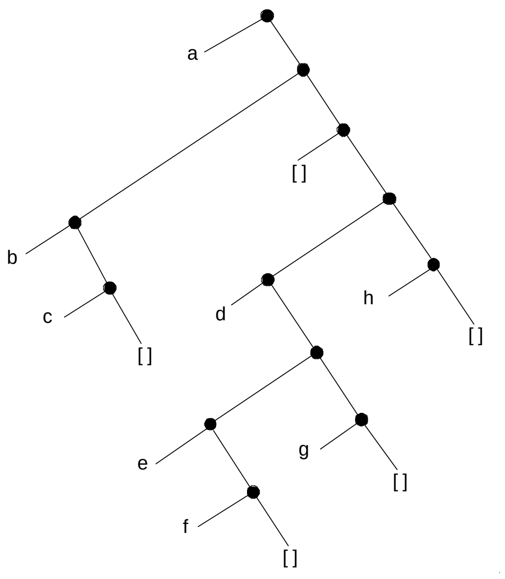


Figure 19: A rotated rendering of the binary tree representation of a list
`[a, [b, c], [ ], [d, [e, f], g], h]`

The binary tree representation corresponds to a typical *internal representation of lists* in languages such as rex, Lisp, and Prolog. The arcs are essentially *pointers*, which are represented as internal memory addresses.

In a way, we have now come a full cycle: we started by showing how trees can be represented as lists, and ended up showing how lists can be represented as trees. The reader may ponder the question: Suppose we start with a tree, and represent it as a list, then represent that list as a binary tree. What is the relationship between the original tree and the binary tree?

Quad-Trees

An interesting and easy-to-understand use of trees is in the digital representation of images. Typically images are displayed based on a two-dimensional array of *pixels* (picture elements), each of which has a black-white, color, or gray-scale value. For now, let us assume black and white. An image will not usually be homogeneous, but will instead consist of regions that are largely black or largely white. Let us use the following 16-by-16 pixel image of a familiar icon as an example:

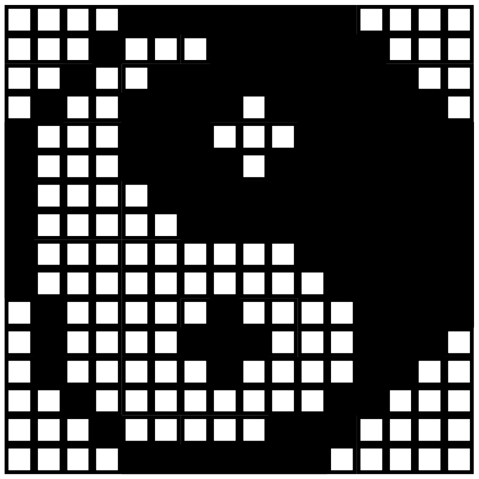


Figure 20: 16 x 16 pixel image of yin-yang icon

The quad-tree is formed by recursively sub-dividing the image into four quadrants, sub-dividing those quadrants into sub-quadrants, and so on, until a quadrant contains only black or only white pixels. The sub-divisions of a region are the four sub-trees of a tree representing the entire region.

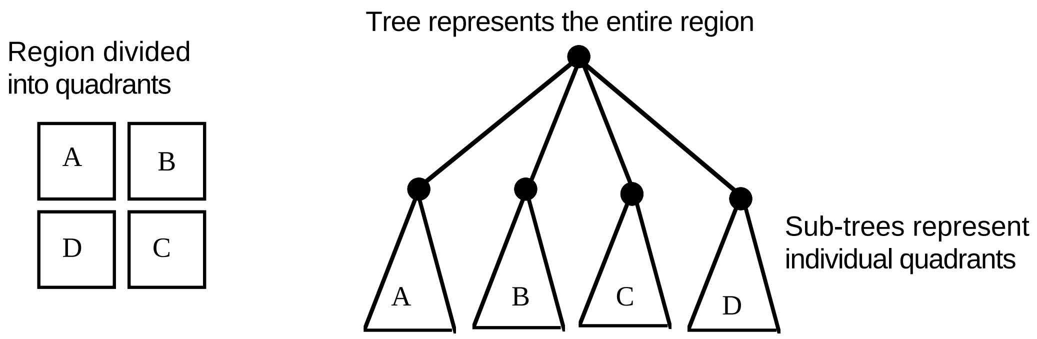


Figure 21: Quad tree recursive sub-division pattern

The first level of sub-division is shown below, and no quadrant satisfies the stopping condition. After the next level sub-division, several sub-quadrants satisfy the stopping condition and will thus be encoded as 0.

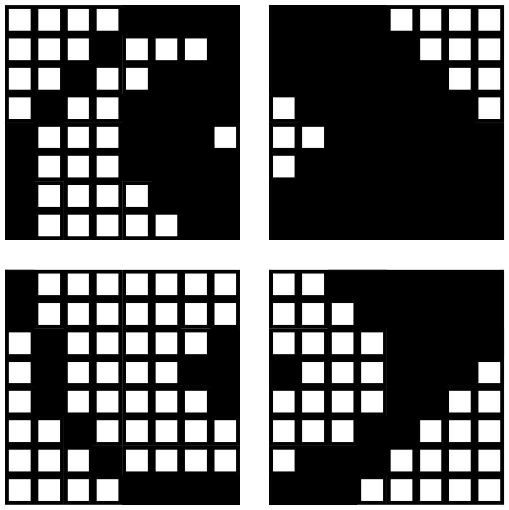


Figure 22: The yin-yang image showing the first level of quad-tree sub-division

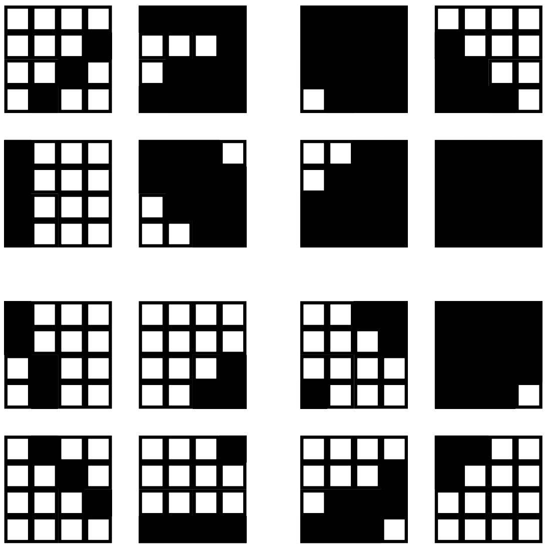


Figure 23: The image after the second level of quad-tree sub-division. Note that one quadrant is all black and will not be further sub-divided.

Continuing in this way, we reach the following set of sub-divisions, in which each quadrant satisfies the stopping condition:

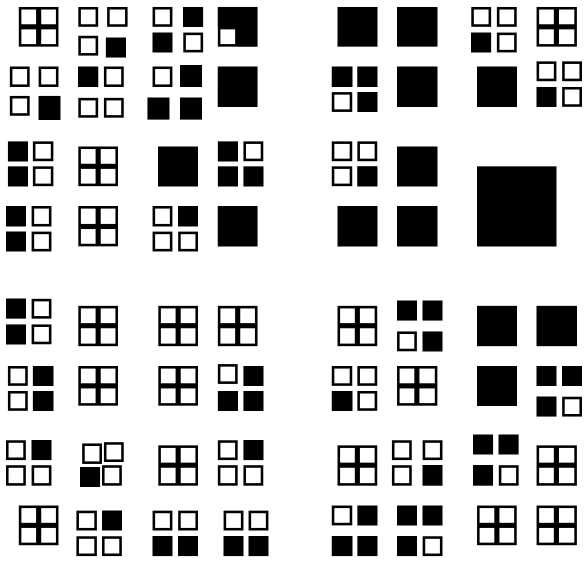


Figure 24: The yin-yang icon sub-divided until all sub-grids are either all black or all white

Encoding the sub-quadrants as 0 for white, 1 for black, we get the quad-tree in Figure 25.

A linear encoding of the quad-tree, using the symbol set { '0', '1', '[', ']' }, reading the quadrants left-to-right and upper-to-lower, is:

```
[[[0[0001][0001][1000]][[0110][1101][0111]1][[1010]0[1010]0][1[1011][01
00]1]][[11[1101]1][[0111]01[0001]][[0001]011]1][[[1010]0[0101]0][000[01
11]][[0100][0010]0[0100]][0[0100][0011][0011]][[0[1101][0111]0][111[111
0]][0[0001][0111][1110]][[1110]000]]]
```

We need to accompany this linear encoding by a single number that indicates the depth of a single pixel in the hierarchy. This is necessary to know the number of pixels in a square area represented by an element in the quad-tree. For example, in the tree above, the depth is 4. An element at the top level (call this level 1) corresponds to a 8-by-8 pixel square. An element at level 2, corresponds to a 4-by-4 area, an element at level 3 to a 2-by-2 area, and an element at level 4 to a single pixel.

The linear quad-tree encoding can be compared in length to the *raster encoding*, that is the array of pixels with all lines concatenated:

```
00001111111100000001000111110000010011111111111000100111011111111010001
11100011111111001111011111111100001111111111110000011111111111000000000
11111110000000000111110100000100001111010000111000111001000001000011000
0100000000110000001000001110000000011111110000
```

Here it is not necessary to know the depth, but instead we need to know the row size. Since both of these are printed in equal-space fonts, based purely on length, modest compression can be seen in the quad-tree encoding for this particular image. In images with larger solid black or solid white areas, a significantly greater compression can be expected.

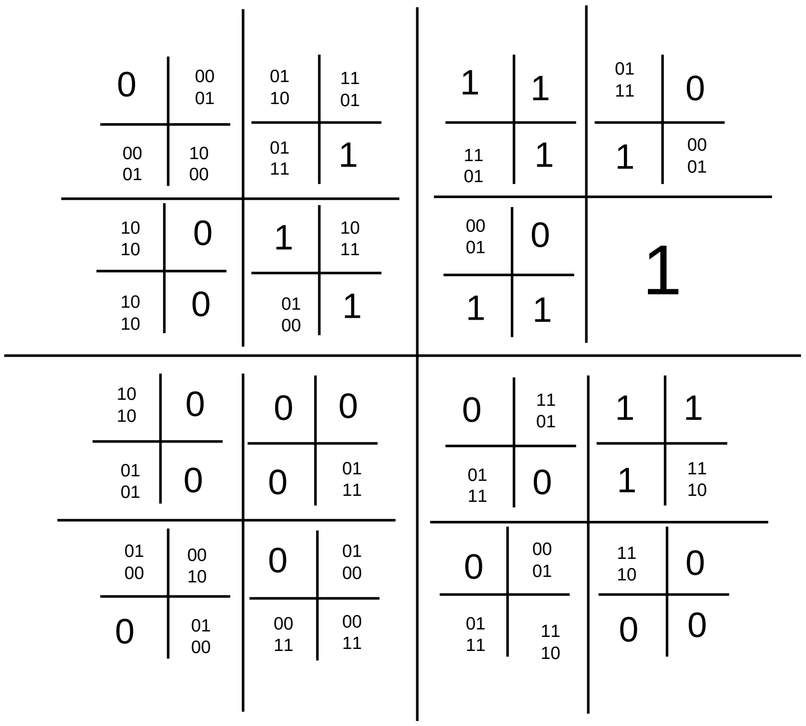


Figure 25: Quad-tree for the yin-yang icon in two dimensions

Quad trees, or their three-dimensional counterpart, oct-trees, have played an important role in algorithm optimization. An example is the well-known Barnes-Hut algorithm for doing simulations of N bodies in space, with gravitational attraction. In this approach, oct-trees are used to structure the problems space so that clusters of bodies can be treated as single bodies in certain computational contexts, significantly reducing the computation time.

Tries

A *trie* is a special kind of tree used for information re*trie*val. It exploits indexability in lists, or arrays, which represent its nodes, in order to achieve more efficient access to the information stored. A trie essentially implements a function from character strings or numerals into an arbitrary domain. For example, consider the function `cities` that gives the list of cities corresponding to a 5-digit ZIP code. Some sample values of this function are:

```
cities(91711) = ["Claremont"]
cities(91712) = [ ]
cities(94305) = ["Stanford", "Palo Alto"]
cities(94701) = ["Berkeley"]
```

The value of `cities` for a given argument could be found by searching a list of the form

```
[
 [91711, "Claremont"],
 [91712],
 [94305, "Stanford", "Palo Alto"],
 [94701, "Berkeley"],
   ....
]
```

(called an *association list* , and elaborated on in the next chapter). However, this would be slower than necessary. A quicker search would result if we used a tree with each non-leaf node having ten children, one for each digit value. The root of the tree would correspond to the first digit of the ZIP code, the second-level nodes would correspond to the second digit, and so on. For any digit combination that cannot be continued to a proper ZIP code, we could terminate the tree construction with a special empty node [] at that point. The overall trie would appear as follows:

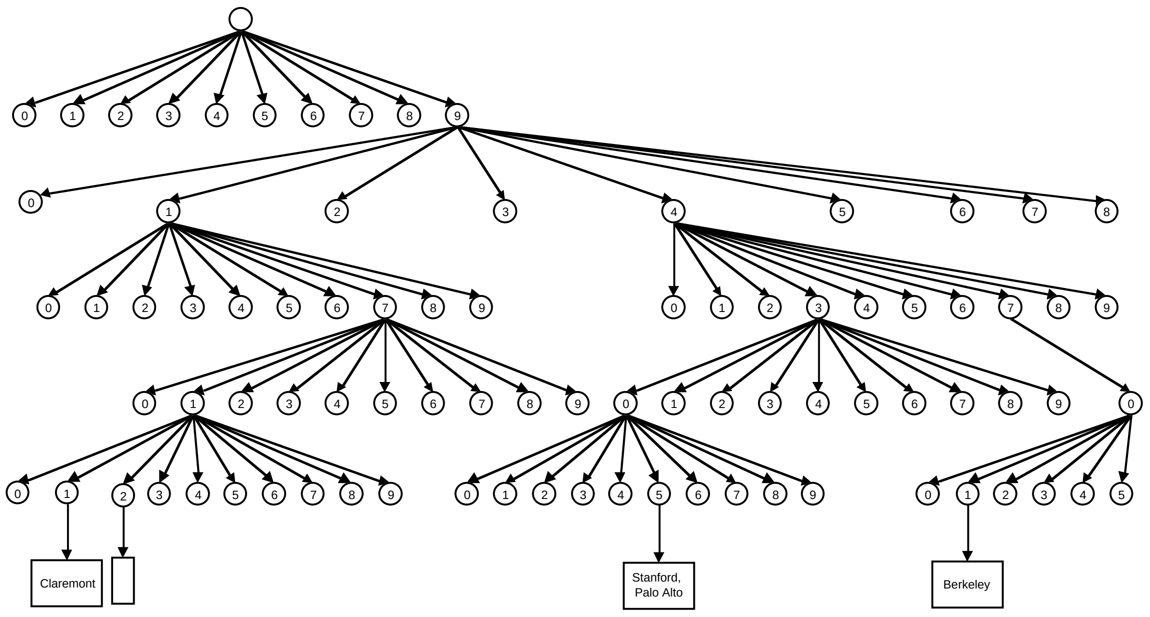


Figure 26: Sketch of part of a trie representing assignment of sets of cities to zip codes.

General Graphs as Lists

If we omit, in the definition of tree, the condition “there is exactly one root”, thus allowing multiple roots (note that there will always be at least one root if the graph is acyclic), we would define the idea of a *forest* or set of trees. In general, taking the root away from a tree will leave a forest, namely those trees with roots that were the targets of the original root.

We can also use the idea of target set to clarify the structure of an *arbitrary* directed graph: Represent the graph as a list of lists. There is exactly one element of the outer list for each node in the graph. That element is a list of the node followed by its targets. For example, Figure 27 shows a directed graph that is not a tree.

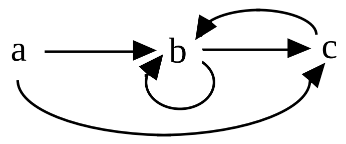


Figure 27: A graph that is not a tree.

The corresponding list representation would be:

```
[ [a, b, c],
  [b, b, c],
  [c, b] ]
```

We have gone through a number of different list representations, but certainly not exhausted the possibilities. It is possible to convert any representation into any other. However, the choice of a working representation will generally depend on the algorithm; some representations lead to simpler algorithms for a given problem than do other representations. The computer scientist needs to be able to:

- Determine the best information representation and algorithm for a given data abstraction and problem.

- Be able to convert from one representation into another and to write programs that do so.

- Be able to present the information abstraction in a means that is friendly to the user. This includes various tabular and graphical forms.

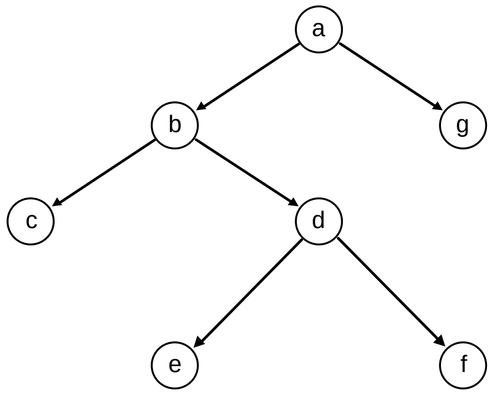
There is no one form that is universally the most correct or desirable. Each application and user community will have its own ways of viewing information and it may even be necessary to provide several views of the same information and the ability of software to switch between them.

One thing is certain: graphs and their attendant information structures are ubiquitous in computer applications. Our entire world can be viewed as sets of intersecting networks of items that connect to one another in interesting ways. The directed graph is the typical way in which a computer scientist would view these connections. To give a modern example, consider the "World-Wide Web". This is an information structure that is distributed over a pervasive network of computers known as the *Internet*. A node in the World-Wide Web is a document containing text and graphics. Each node has a symbolic *address* that uniquely identifies it among all other documents. Typically a document will refer to one or more other documents in the form of *hypertext links*. These links specify the addresses of other nodes. Thus they correspond directly to the arcs in a directed graph. When the user clicks on the textual representation of a link using a *web browser* the browser changes its focus to the target of that particular link.

The World-Wide Web is thus a very large directed graph. It is not a tree, since it is cyclic: nodes can refer to each other or directly to themselves. Moreover, it is common for the target sets of two nodes to have a non-empty intersection. Finally there is no unique root. Thus all three of the requirements for a directed graph to be a tree are violated.

Exercises

1. •• What are some ways of representing the following tree as a list:



2. •• Draw the trees corresponding to the interpretation of the following list as (i) a labeled tree, (ii) an unlabeled tree, and (iii) a binary tree:

```
[1, [2, 3, 4, 5], [6, [7, [8, 9] ] ] ]
```

3. ••• What lists have interpretations as unlabeled trees but not as labeled trees? Do these lists have interpretations as binary trees?

4. ••• Identify some fields outside of computer science where directed graphs are used. Give a detailed example in each case.

5. •• The decoding of the *Morse code* alphabet can be naturally specified using a trie. Each character is a sequence of one of two symbols: dash or dot. For example, an 'a' is dot-dash, 'b' is dash-dot-dot-dot, and so on. Consult a source, such as a dictionary, which contains a listing of the Morse code, then construct a decoding trie for Morse code.

6. ••• A *tune identifier* based on the trie concept can be developed to provide a simple means of identifying unknown musical tunes. With each tune we may associate a signature that describes the pattern of intervals between the initial notes, say the first ten, in the tune. The symbol D indicates that the interval is a downward one, U indicates it is upward, and S represents that the note is the same. For example, the tune of *The Star Spangled Banner* has the signature DDUUU UDDDU, corresponding to its start:

```
O - oh say can you see, by the dawn's ear-ly
  D   D   U   U   U    U  D   D      D   U
```

   Some other tunes and their signatures are:

```
Bicycle Built for Two                       DDDUU UDUDU
Honeysuckle Rose                            DDUUU DDUUU
California Here I come                      SSSUD DDSSS
One-Note Samba                              SSSSS SSSSS
```

   Show how a trie can be used to organize the songs by their signatures.

7. ••• The three-dimensional counterpart of a pixel is called a *voxel* (for volume element). Describe how a 3-dimensional array of voxels can be compacted using the idea of an *oct-tree* (eight-way tree).

8. •• An *oriented directed graph* is like a directed graph, except that the targets have a specific order, rather than just being a set. Describe a way to represent an oriented directed graph using lists.

9. ••• A *labeled directed graph* is a directed graph with labels on its arrows. Describe a way to represent a labeled directed graph using lists.

10. •• Draw enough nodes of the World-Wide Web to show that it violates all three of the tree requirements.

2.8 Matrices

The term *matrix* is often used to refer to a table of two dimensions. Such a table can be represented by a list of lists, with the property that all elements of the outer list have the same length.

One possible use of matrices is as yet another representation for directed graphs. In this case, the matrix is called a *connection matrix* (sometimes called *adjacency matrix*) and the elements in the matrix are 0 or 1. The rows and columns of the matrix are indexed by the nodes in the graph. There is a 1 in the i*th* row j*th* column of the matrix exactly when there is a connection from node i to node j. For example, consider the graph below, which was discussed earlier:

$$\begin{tikzcd} a \arrow[r] \arrow[rr, bend right] & b \arrow[r] \arrow[loop below] & c \arrow[l, bend right] \end{tikzcd}$$

The connection matrix for this graph would be:

$$\begin{array}{c|ccc} & \mathbf{a} & \mathbf{b} & \mathbf{c} \\ \hline \mathbf{a} & 0 & 1 & 1 \\ \mathbf{b} & 0 & 1 & 1 \\ \mathbf{c} & 0 & 1 & 0 \end{array}$$

Note that there is one entry in the connection matrix for every arrow. We can use the connection matrix to quickly identify nodes with certain properties:

A root corresponds to a column of the matrix with no 1's.

A leaf corresponds to a row of the matrix with no 1's.

As a list of lists, the connection matrix would be represented

```
[ [0, 1, 1],
  [0, 1, 1],
  [0, 1, 0] ]
```

A related matrix is the *reachability matrix*. Here there is a 1 in the i*th* row j*th* column of the matrix exactly when there is a directed *path* (one following the direction of the arrows) from node i to node j. For the current example, the reachability matrix would be

$$\begin{array}{c|ccc} & \mathbf{a} & \mathbf{b} & \mathbf{c} \\ \hline \mathbf{a} & 0 & 1 & 1 \\ \mathbf{b} & 0 & 1 & 1 \\ \mathbf{c} & 0 & 1 & 1 \end{array}$$

From the reachability matrix, we can easily determine if the graph is cyclic: It is iff there is a 1 on the (upper-left to lower-right) diagonal, indicating that there is a path from some node back to itself. In the present example, both rows b and c have 1's in their diagonal elements.

Notice that the reachability matrix, since it does represent a set of pairs, also corresponds to a relation in its own right. This relation is called the *transitive closure* of the original relation. We typically avoid drawing graphs of transitive closures because they tend to have a lot more arrows than does the original graph. For example, the transitive closure of this simple graph

$$\begin{tikzcd}
a \arrow[r] & b \arrow[r] & c \arrow[r] & d
\end{tikzcd}$$

would be the more complex graph

$$\begin{tikzcd}
a \arrow[r] \arrow[rrr, bend left] \arrow[rr, bend right] & b \arrow[r] \arrow[rr, bend right] & c \arrow[r] & d
\end{tikzcd}$$

As a second example of a matrix, we could use a table of 0's and 1's to represent a black-and-white *image*. Of course, the entries in a matrix aren't limited to being 0's and 1's. They could be numeric or even symbolic expressions.

Apparently we can also represent tables of *higher dimension* using the same list approach. For every dimension added, there would be another level of nesting.

Exercises

1. • Show a list of lists that does not represent a matrix.

2. •• Suppose a binary relation is given as the following list of pairs. Draw the graph. Construct its connection matrix and its reachability matrix. What is the list of pairs corresponding to its transitive closure?

    ```
    [ [1, 2], [2, 3], [2, 4], [5, 3], [6, 5], [4, 6] ]
    ```

3. ••• Devise a method, using connection matrices, for determining whether a graph is acyclic.

4. •• Consider the notion of a labeled directed graph as introduced earlier (a graph with labels on its arrows). How could you use a matrix to represent a labeled directed graph? How would the representation of this matrix as a list differ from previous representations of graphs as lists?

5. ••• Devise a method for computing the connection matrix of the transitive closure of a graph, given the graph's connection matrix.

2.9 Undirected graphs

An *undirected graph* is a special case of a directed graph (and not the other way around, as you might first suspect). An undirected graph is usually presented as a set of nodes with lines connecting selected nodes, but with no arrow-heads on the lines. In terms of a directed graph, i.e. a set of pairs of nodes, a line of an undirected graph connecting a node `n` to a node `m` represents *two* arrows, one from node `n` to `m` and another from node `m` to `n`. Thus the connection matrix for an undirected graph is always *symmetric* about the diagonal (that is, if there is a 1 in row i column j, then there is a 1 in row j column i).

On the left below is an undirected graph, and on the right the corresponding directed graph.

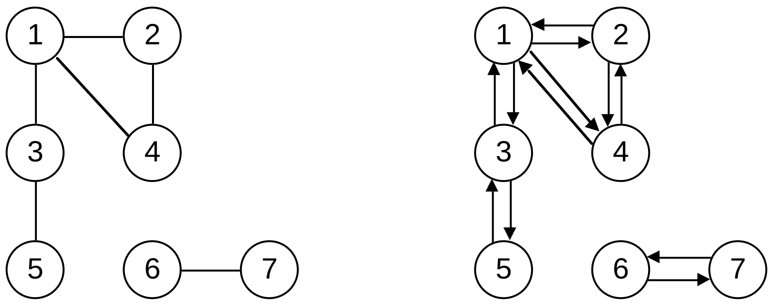


Figure 28: An undirected graph and its corresponding directed graph

In summary, an undirected graph may be treated as an abbreviation for a directed graph. All the representational ideas for directed graphs thus apply to undirected graphs as well.

Exercises

1. • Show the list of pairs representing the undirected graph above. Show its connection matrix.

2. •• Devise an algorithm for determining whether a directed graph could be represented as an undirected graph.

3. ••• The transitive closure of an undirected graph has an unusual form. What is it? Devise a more efficient way to represent such a transitive closure as a list based on its properties.

2.10 Indexing Lists vs. Arrays

We will *index* elements of a list by numbering them starting from 0. Given a list and index, an element of the list is determined uniquely. For example, in

```
[“Canada”, “China”, “United Kingdom”, “Venezuela”]
```

the index of `“China”` is 1 and that of `“Venezuela”` is 3. The important thing to note is that indices are not an explicit part of the data. Instead, they are implicit: the item is determined by “counting” from the beginning.

In rex, an item at a specific index in the list may be determined by treating that list as if it were a function: For example

```
rex > L = ["Canada", "China", "United Kingdom", "Venezuela"];
1

rex > L(2);
United Kingdom
```

Indexing lists in this way is not something that should be done routinely; for routine indexing, *arrays* are a better choice. This is now getting into representational issues somewhat, so we only give a preview discussion here.

The time taken to index a *linked list* is proportional to the index.

On the other hand, due to the way in which computer memory is constructed, the time taken to index an array is nearly constant. We refer to the use of this fact as the *linear addressing principle*.

linear addressing principle:

The time taken to index an *array* is constant, independent of the value of the index.

We use the modifier "linear" to suggest that we get the constant-time access only when elements can be indexed by successive integer values. We cannot expect constant-time access to materialize if we were to instead index lists by arbitrary values. (However, as will be seen in a later chapter, there are ways to come close to constant-time access using an idea known as *hashing*, which maps arbitrary values to integers.)

In addition to lists, arrays are available in rex. For example, a list may be converted to an array and vice-versa. Both are implementations of the abstraction of *sequence*, differing primarily in their performance characteristics. Both can be indexed as shown above. We will discuss this further in the next chapter.

2.11 Structure Sharing

Consider lists in which certain large items are repeated several times, such as

```
[ [1, 2, 3], 4, [1, 2, 3], 5, [1, 2, 3], 6]
```

In the computer we might like to normalize this structure so that space doesn't have to be taken representing the repeated structure multiple times. This is especially pertinent if we do not tend to modify any of the instances of the recurring list. Were we to do that, we'd have to make a decision as to whether we wanted the structures to be shared or not.

A way to get this kind of sharing in rex is to first bind an identifier to the item to be shared, then use the identifier in the outer structure thus:

```
rex > shared = [1, 2, 3];
1

rex > outer = [shared, 4, shared, 5, shared, 6];
1

rex > outer;
[[1, 2, 3], 4, [1, 2, 3], 5, [1, 2, 3], 6]
```

Within the computer, sharing is accomplished by *references*. That is, in the place where the shared structure is to appear is a data item that *refers* to the *apparent* item, rather than being the item itself. From the user's viewpoint, it appears that the item itself is where the reference is found. In rex, there is no way to tell the difference.

A reference is usually implemented using the lower-level notion of a *pointer*. Pointers are based on another use of the linear addressing principle: all items in memory are stored in what amounts to a very large array. This array is not visible in a functional language such as rex, but it is in languages such as C. A pointer or reference is an index into this large array, so we can locate an item given its reference in nearly constant time.

In programming languages, references typically differ from pointers in the following way: a pointer must be prefixed or suffixed with a special symbol (such as prefixed with a * in C or suffixed with a ^ in Pascal) in order to get the value being referenced. With references, no special symbol is used; references *stand for* the data item to which they refer. Most functional languages, as well as Java, have no pointers, although they may have references, either explicit or implicit.

In addition to sharing items in lists, *tails* of lists can be shared. For example, consider

```
rex > tail = [2, 3, 4, 5];
1

rex > x = [0 | tail];
1

rex > y = [1 | tail];
```

```
1

rex > x;
[0, 2, 3, 4, 5]

rex > y;
[1, 2, 3, 4, 5]
```

Although x and y appear to be distinct lists, they are sharing the list tail as a common *rest*. This economizes on the amount of storage required to represent the lists.

As we saw earlier, lists have a representation as unlabeled trees. However, a tree representation is not appropriate for indicating sharing. For this purpose, a generalization of a tree called a *dag* (for *directed acyclic graph*) helps us visualize the sharing.

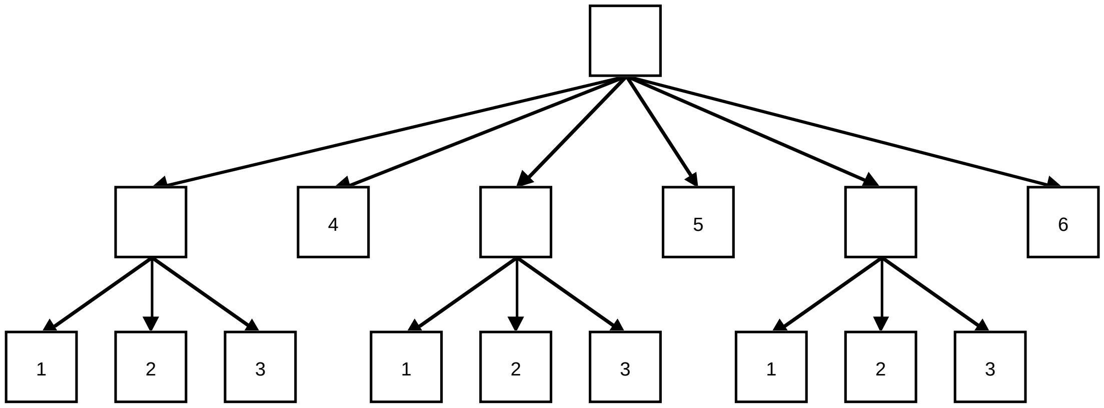


Figure 29: A tree with sharable sub-trees.

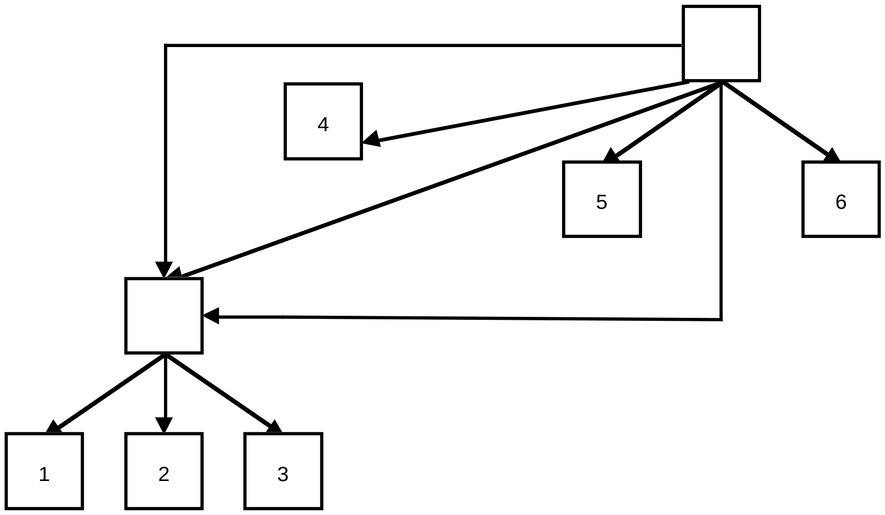
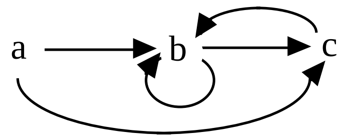


Figure 30: Showing shared structure using a dag.

Strictly speaking, the above tree and dag are not simply directed graphs, but instead directed *oriented* graphs, meaning that *ordering* and *repetition* of targets is important. If it were purely a graph, then the targets would be regarded as a set and repetitions wouldn't matter, defeating our use of trees. These fine distinctions will not be belabored, since we are using lists to represent them anyway and we are trying to avoid building up a vocabulary with too many nuances.

Graphs as well as dags can be represented using *references*. This results in a more compact representation. In order to demonstrate such a use of references in rex, we will introduce an identifier for each node in the graph. The identifier will be bound to a list consisting of the name of the node followed by the values of the identifiers of the targets of the node. Consider for example the graph introduced earlier:



We would like to define the three node lists simultaneously. To a first approximation, this would be done by the rex statement:

```
[a, b, c] = [ ["a", b, c], ["b", b, c], ["c", b] ];
```

There is a problem with this however; the right-hand side is evaluated *before* binding the variables on the left-hand side, but the symbols `a`, `b`, and `c` on the right are not yet defined. The way to get around this is to *defer* the use of those right-hand side symbols. In general, *deferred binding* means that we use a symbol to designate a value, but give its value later. Deferring is done in rex by putting a `$` in front of the expression to be deferred, as in:

```
rex > a = ["x", $b, $c];
1

rex > b = ["y", $b, $c];
1

rex > c = ["z", $b];
1
```

While these equations adequately define a graph information structure inside the computer, there is a problem if we try to print this structure. The printing mechanism tries to display the list structure as if it represented a tree. If the graph above is interpreted a tree, the tree would be infinite, since node `a` connects to `b`, which connects to `b`, which connects to `b`, ... Our rex interpreter behaves as follows:

```
rex > a;
[x, [y, [y, [y, [y, [y, [y, [y, [y, [y, [y, ...
```

where the sequence continues until terminated from by the user. We would need to introduce another output scheme to handle such cycles, but prefer to defer this issue at the moment.

Exercises

1. •• Describe how sharing can be useful for storing quad trees.

2. ••Describe how sharing can be useful for storing matrices.

3. ••• Describe some practical uses for deferred binding.

4. ••• Devise a notation for representing an information structure with sharing and possible cycles.

5. ••• For a computer operating system of your choice, determine whether and how sharing of files is represented in the directory structure. For example, in UNIX there are two kinds of *links*, regular and symbolic, which are available. Describe how such links work from the user's viewpoint.

2.12 Abstraction, Representation, and Presentation

We have used terms such as "abstraction" and "representation" quite freely in the foregoing sections. Having exposed some examples, it is now time to clarify the distinctions.

- By information *presentation*, we mean the way in which the information is presented to the user, or in which the user presents information to a computational system.

- By information *representation*, we mean the scheme or method used to record and manipulate the information, such as by a list, a matrix, function, etc., for example, inside a computer.

- By information *abstraction*, we mean the intangible set of "behaviors" that are determined by the information when accessed by certain functions and procedures.

A presentation can thus be viewed as a representation oriented toward a user's taste.

Number Representations

As a very simple example to clarify these distinctions, let us consider the domain of natural numbers, or "counting numbers", which we normally denote 0, 1, 2, 3, ... When we write down things that we say represent numbers, we are dealing with the *presentation* of the numbers. We do this in using *numerals*. For example, 5280 would typically be understood as a decimal numeral. The characters that make up a numeral are called *digits*. The actual *number* itself is determined by *abstract* properties. For example, the number of "ticks" we'd need to perform in counting starting from 0 to arrive at this number determines the number exactly and uniquely. There is only one number requiring exactly that many actions. Alternatively, the number of times we can remove a counter from a number before reaching no counters also determines the number.

Regarding representation of numbers in the computer, typically this would be in the form of a sequence of electrical voltages or magnetic flux values, which we abstract to *bits*: 0 and 1. This is the so-called *binary representation* (not to be confused with binary relations). Each bit in a binary representation represents a specific power of 2 and the number itself is derived by summing the various powers of two. For example, the number presented as decimal numeral 37 typically would be represented as the sequence of bits

1 0 0 1 0 1

meaning

$$\mathbf{1} * 2^5 + \mathbf{0} * 2^4 + \mathbf{0} * 2^3 + \mathbf{1} * 2^2 + \mathbf{0} * 2^1 + \mathbf{1} * 2^0$$

i.e.

$$\mathbf{32} \quad + 0 \quad + 0 \quad + \mathbf{4} \quad + 0 \quad + \mathbf{1}$$

The appeal of the binary representation is that it is complete: every natural number can be represented in this way, and it is simple: a choice of one of two values for each power of two is all that is necessary.

An enumeration of the binary numerals for successive natural numbers reveals a pleasing pattern:

```
0          0
1          1
2         10
3         11
4        100
5        101
6        110
7        111
8       1000
    ...
```

The pattern is: if we look down the rightmost column, we see that 0's and 1's alternate on every line. If we look down the second rightmost column, they alternate every two lines, in the third every four lines, etc. In order to find the least-significant (rightmost) digit of the binary representation, we can divide the number by 2 (using whatever representation we want). The *remainder* of that division is the digit. The reader is invited, in the exercises, to extend this idea to a general method. In the programming languages we use, `N % M` denotes the remainder of dividing `N` by `M`.

Most real applications deal with information abstractions far more complex than just the natural numbers. It is important for the computer scientist to be familiar with typical abstractions and their attendant presentations and representations and to be able to spot instances of these abstractions as they arise in problems. This is one of the reasons that we have avoided focusing on particular common programming languages thus far; it is too easy to get lost in the representational issues of the language itself and loose sight of the information structural properties that are an essential component of our goal: to build well-engineered information processing systems.

Sparse Array and Matrix Representations

As another example of representation vs. abstraction, consider the abstract notion of arrays as a sequence indexed by a range of integers 0, 1, 2, ... N. A sparse array is one in which most of the elements have the same value, say 0. For example, the following array might be considered sparse:

```
[0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
```

since most of the elements are `0`. We could represent this array in a more compact, and in some ways more readable, as a list of only the non-zero elements and the values of those elements:

```
[ [1, 1], [3, 1], [7, 1], [9, 1], [13, 1], [27, 1] ]
```

Here the first element of each pair is the index of the element in the original array.

Likewise, we can consider a matrix to be an array indexed by a pair of values: row and column. The following sparse matrix (where blank entries imply to be some fixed value, such as 0)

0							
				5			
							3
					6		
		2					
						4	
	7						
			1				

could be represented as a list of non-zero elements of the form [row, col, value] (recalling that row and column indices start at 0):

```
[ [0, 0, 0], [6, 1, 7],
  [4, 2, 2], [7, 3, 1],
  [1, 4, 5], [3, 5, 6],
  [5, 6, 4], [2, 7, 3]]
```

This representation required 24 integers, rather than the 100 that would be required for the original matrix. While sparse array and matrix representations make efficient use of memory space, they have a drawback: linear addressing can no longer be used to access their elements. For example, when a matrix is to be accessed in a columnar fashion, we may have to scan the entire list to find the next element in a given column. This particular problem can be alleviated by treating the non-zero elements as nodes in a pair of directed graphs, one graph representing the columns and the other representing the rows, then using pointer or reference representations to get from one node to another.

Other Data Presentations

As far as different presentations, the reader is probably aware that there are many different presentations of numbers, for example different bases, Arabic vs. Roman numerals, etc. The same is true for lists. We have used one presentation of lists. But some languages use *S expressions*, ("S" originally stood for "symbolic"), which are similar to ours but with rounded parentheses rather than square brackets and omitting the commas. To differentiate, we could call ours *R expressions* (for rex expressions).

R expression: `[1, [2, 3], [ [ ], 4] ]`

S expression: `(1 (2 3) (() 4))`

S expressions often use entirely the same representation in the computer that we use: singly-linked lists. S expressions are used in languages Lisp and Scheme, while R expressions are used in the language Prolog. In Lisp and Scheme, programs, as well as data, are S expressions. This uniformity is convenient when programs need to be manipulated as data.

Regarding representations, we do not regard singly-linked lists as universally the most desirable representation. They are a simple representation and in many, but not all, cases can be a very efficient one. We will look at this and other alternatives in more detail in later chapters.

In our view, it is too early to become complacent about the abstractions supported by a given language. There are very popular languages on the scene that do a relatively poor job of supporting many important abstractions. We are not proposing rex as a solution, but only as a source of ideas, some of which some of our readers hopefully will design into the next generation of languages.

Exercises

1. •• Enumerate the numbers from 9 to 31 as binary numerals.

2. ••• Devise a method for converting a number into its binary representation.

3. ••• Devise a method for converting a binary numeral into decimal.

4. •• Devise a method for transposing a matrix stored in the sparse representation described in this section. Here *transpose* means to make the columns become rows and vice-versa.

5. •• A *formal polynomial* is an expression of the form

$$a_0 + a_1x + a_2x^2 + a_3x^3 + ... + a_{N-1}x^{N-1}$$

While polynomials are often used as representations for certain functions, the role played by a formal polynomial is just to represent the sequence of coefficients

$$[a_0, a_1, a_2, a_3, ..., a_{N-1}]$$

The "variable" or "indeterminate" x is merely a symbol used to make the whole thing readable. This is particularly useful when the sequence is "sparse" in the sense that most of the coefficients are 0. For example, it would not be particularly readable to spell out all of the terms in the sequence represented by the formal polynomial

$$1 + x^{100}$$

Devise a way to represent formal polynomials using lists, without requiring the indeterminate symbol.

2.13 Abstract Information Structures

Previously in this chapter, we exposed a number of different information structures. A common characteristic of the structures presented was that we could *visualize* the structure, using presentations such as lists. But there is another common way to characterize structures and that is in terms of *behavior*. Information structures viewed this way are often called *abstract data types* (ADTs). This approach is the essence of an important set of ideas called *object-oriented programming*, which will be considered later from an implementation point of view. An object's class can be used to define an ADT, as the class serves to group together all of the operators for a particular data type.

As an example, consider the characterization of an array. The typical behavioral aspects of arrays are these:

- We can make an array of a specified length.

- We can set the value of the element at any specified index of the array.

- We can get the value of the element at any specified index.

These are the things that principally characterize an array, although we can always add others, such as ones that search an array, search from a given index, search in reverse, etc. All of these aspects can be understood carried out in terms of the three basic behaviors given, without actually "seeing" the array and how it is represented. We can postulate various abstract operators to represent the actions:

`makeArray(N)`	returns an array of size `N`
`set(A, i, V)`	sets the element of array `A` at index `i` to value `V`
`get(A, i)`	returns the element of array `A` at index `i`
`size(A)`	returns the size of array `A`

Note that aspects of *performance* (computation time) are not shown in these methods. While it would be efficient to use the linear addressing principle to implement the array, it is not required. We could use a sparse representation instead. What is implied, however, is that there are certain relationships among the arguments and return values of the methods. The essence is characterized by the following relationships, for any array `A`, with .... representing a series of zero or more method calls:

A sequence

```
A = makeArray(N); .... size(A)
```

returns `N`, the size of the array created.

A sequence

```
set(A, i, V); .... get(A, i)
```

returns `V`, provided that within .... there is no occurrence of `set(A, i, V)` and provided that `i > 0` and `i < size(A)`.

These summarize an *abstract array*, independent of any "concrete" array (*concrete* meaning a specific implementation, rather than an abstraction).

This kind of thinking should be undertaken whenever building an abstraction. That is, begin by asking the question

What are the properties that connect the operators?

The answers to such questions are what guide the choice or development of implementations, which will be the subject of much of this book.

Exercises

1. ••• Express the idea of a *list* such as we constructed using [.... | ....] from an abstract point of view, along the lines of how we described an abstract array.

2. •••• A stack is an abstraction with a behavior informally described as follows: A stack is a repository for data items. When the stack is first created, the stack contains no items. Item are inserted one at a time using the *push* method and removed one at a time using the *pop* method. The stack determines which element will be removed when pop is called: elements will be removed in order of most recently inserted first and earliest inserted last. (This is called a LIFO, or last-in-first-out ordering). The *empty* method tells whether or not the stack is empty. Specify the relationships among methods for a stack.

3. •• Many languages include a *struct* facility for structuring data. (An alternate term for struct is *record*.) A struct consists of a fixed number of components of possibly heterogenoeous type, called *fields*. Each component is referred to by a name. Show how structs can be represented by lists.

2.14 Conclusion

In this chapter, we have primarily exposed ways of thinking about information structures. We have not written much code yet, and have said little about the implementations of these structures. We mentioned in the previous section the idea of abstract data types and the related idea of "objects". Objects and information structures such as lists should not be viewed as exclusive. Instead, there is much possible synergism. In some cases, we will want to use explicit information structures containing objects as their elements, and in others we will want to use information structures to implement the behavior defined by classes of objects. It would be unfortunate to box ourselves into a single programming language paradigm at this point.

2.15 Chapter Review

Define the following concepts or terms:

acyclic
ADT (abstract data type)
array
binary relation
binary representation of numbers
binary tree representation of list
binding
connection matrix
dag
deferred binding
directed graph
forest
functional programming
index
information sharing
labeled-tree interpretation of a list

leaf
leafcount
length (of a list)
linear addressing principle
list
list equality
list matching
numeral vs. number
quad tree
queue
R expression
reachable
reachability matrix
reference
root
S expression
sharing
sparse array
stack
target set
transitive closure
tree
trie
unlabeled-tree interpretation of a list